A Multitasking Kernel under MSDOS

by

Y. S. S. KRISHNA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
March 1994

A Multitasking Kernel under MSDOS

A thesis submitted
in partial fulfilment of the requirements
for the degree of

Master of Technology

by

Y.S.S. Krishna

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

March, 1994
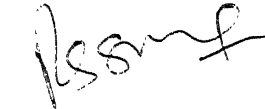
CERTIFICATE

It is certified that the work contained in the thesis titled *A Multitasking Kernel under MSDOS*, by Y.S.S. Krishna has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

March, 1994

Dr. Rajat Moona
Assistant Professor,
Department of Computer Science
and Engineering,
IIT Kanpur

ACKNOWLEDGEMENTS

Abstract

This report describes the design and implementation of multitasking on MS-DOS operating system. The platform is taken as 80386 based PC-ATs. The processing speed of PC-386ATs is comparable with that of workstations, while they are comparatively cheaper. This has motivated us to upgrade the MS-DOS operating system running on PC-386ATs so that UNIX based software can be ported to the PC environment.

In our implementation we have implemented the multitasking in the form of *fork* system call as in the case of UNIX. The other related system calls for supporting the multitasking are also implemented.

Contents

List of Figures

Chapter 1

Introduction

1.1 Current state of PC software and hardware

The growing availability and advancement of the microprocessors have made the existing software sophistication all the more important. The Personnel Computers (PCs) which are based on microprocessors have made the computing power very cheap. But the advancement of the microprocessor hardware technology has created a big technological gap between the software which is being used for the PCs, and the hardware.

The hardware technological growth of PCs have narrowed the speed difference between Workstations and PCs. This has led to the idea of using the PCs as Workstations as they are comparatively cheaper. For using the PCs as Workstations we have to port the software which is being used on the Workstations to the PCs. This porting is not very easy since the operating system platforms on the Workstations are far more advanced than the ones used for PCs. Porting a new operating systems to the PCs renders the old applications obsolete. Hence the PC level operating systems have to be improvised by adding the new features while ensuring compatibility with old applications.

In general an UNIX like operating system running on Workstations supports multitasking, networking, user level authentication etc. These features are generally absent in the operating systems running on PCs. One such example is MS-DOS.

1.2 MS-DOS operating system

The MS-DOS is a single task operating system, and it was originally designed to run on 8088 based PC-XTs. The operating system is highly machine dependent and suffers several limitations some of which are outlined below.

1. The machine was intended to be for personnel use and therefore the operating system does not provide any user level authentication. Thus a malicious user can harm the file system or any other resource. His actions may even lead to the crashing of the system.

2. The operating system is not reentrant. Thus if the operating system calls a user level routine this routine can not use MS-DOS services. This restricts MS-DOS from being shared among different user level tasks.

3. The entire user address space is limited to 1Mb. This includes the space for the operating system, ROM BIOS and the user programs. In later versions of MS-DOS, the extended memory beyond 1Mb is supported by the operating system. However in such cases the presence and usage of this memory are not transparent to the application.

4. The system resources are not protected by the MS-DOS, making them visible to the user programs. Thus a user program may read/write data from any input/output port of the system.

5. It does not support any built in network driver. Thus to do any network related operations, a user program needs a network driver.

1.3 Motivation for the present work

As we have discussed above the MS-DOS operating system was developed for 8088 microprocessor based systems. But now the higher performance processors like 80286, 80386, 80486 etc based systems are being used for PCs. The operating system used for these advanced processors based PCs is also MS-DOS to retain the compatibility with old applications. This has motivated us to improvise MS-DOS to make it flexible enough to incorporate the new features of these high level processors and add extra feature to make it work as a true multitasking operating system. As we do not have the source code of MS-DOS, we could

not make it work as a truly multitasking operating system in terms of efficient memory management.

We have chosen the 80386 microprocessor based systems for our work. The 80386 microprocessor is 32 bit wide, supports paging and lot of other features which are explained in the next chapter. We have implemented multitasking in a manner similar to that in UNIX operating system. We have added extra system calls to the MS-DOS int 21h functions. The detailed list is given in the appendix.

This is the initial work to make the MS-DOS applications take advantage of the UNIX based applications like X Windows, inctd etc so that 80386 based systems can be used as Workstations.

1.4   Organization of the thesis

The organization of this thesis in the remaining chapters is as follows,

In Chapter 2 we discuss the features of 80386 microprocessor which make it usable for our work. In Chapter 3 we give the detailed design of our work. Chapter 4 is the implementation of the multitasking. Chapter 5 concludes this thesis and provides future extensions to this work.

Chapter 2

Support provided by 80386 processor

2.1   Overview of 80386 processor

The 80386 [Tutl88] processor provides the following new features which were absent in the earlier processors of the iAPX 86 family.

- 32-bit programmer's register set

- 32-bit data bus

- 32-bit address bus

- New Instructions

- Support for multitasking and paging

The most important new feature is the addition of a full demand-paged memory management unit (MMU). The MMU allows the systems programmer to take a whole new approach to operating system design. This MMU allows the microprocessor based computer to have the capabilities normally possible with a minicomputer. The general large scale functions such as multitasking, demand paging, address translation, and virtual memory are all supported by the 80386 processor.

Compared to the earlier processors of iAPX 86 family 80386 processor has more than a dozen registers. Some of these extra registers are used for memory management.

Like all its predecessors, the 80386 maintains object code (binary level) compatibility with previous members of its line. The 80386 is similar enough to the 80286 that compatibility with it was relatively easy to achieve. However, the 80386 has to enter a special *8086 mode* to execute 8086 programs in protected mode. This "split personality" feature, along with the 80386's inherent multitasking ability, allows us to run 8086 code alongside 80286 or 80386 programs or even to run multiple 8086 programs simultaneously.

2.2   Operating modes

The modes, in order of increasing complexity, are Real Address mode, Virtual 8086 mode, and Protected Virtual Address mode.

2.2.1   Real Address mode (Real mode)

Real Address mode, or simply Real mode, is the simplest and most basic operating mode of 80386. It is much like an 8086 style of operation. In this mode user program has the the highest privilege level and can use all privileged instructions of 80386 which may be required to take the system to protected mode of operation.

The 80386 processor powers up in Real mode. It is a greatly simplified operating mode that we can use while we prepare for the transition into one of the other two modes (usually Protected mode).

Of the 80386's three major operating modes, Real mode most closely approximates an actual 8086/88. This is the mode in which most 80386 based PC/ATs run today. The MS-DOS operating system was never designed to deal with flexible segmentation, privilege protection, multitasking, paging, or any of the other features available in Protected mode or even in Virtual 8086 mode, and so Real mode is the easiest solution. But if we have to make the MS-DOS a multitasking operating system we have to run the MS-DOS in the Virtual 8086 mode.

2.2.2   Protected Virtual Address mode (Protected mode)

In this mode all capabilities of 80386 are used. Addressing in this mode has an entirely different mechanism compared to the one in 8086, which is based on segmentation. The

code here can run in both 16bit and 32bit addressing modes. The processor allows four levels of protection, and paging in this mode.

The MS-DOS applications can not be made to run in this mode as this mode supports a different addressing mechanism and exception handling.

2.2.3   Virtual 8086 mode (VM86 mode)

The processor in this mode behaves exactly like an 8086/88 processor. The exception handling is however very different from that in 8086/88. The privilege level of the code running in this mode is 3 i.e the lowest privilege. This mode can be entered through a task switch. In this mode paging can be active. The virtual address space of the task running in this mode is same as the physical memory supported by 8086/88 processor i.e 1Mb.

2.3   Multitasking in 80386

2.3.1   80386 Task state segments

On the 80386, the basic unit of a multitasking system is the task. The definition of task in our system defined in later chapters is different from the definition given here. A task can be a single program, or it can be a group of related programs. If the 80386 system is to have multiple users, a task can be assigned to each user task (the user task implies the "task" we mentioned in earlier paragraphs).

From the point of view of the 80386, the definition of a task is very clear. A task is any collection of code and data and has a Task State Segment (TSS) assigned to it. There is a one-to-one correlation between TSSs and tasks. The TSS is the 80386 equivalent of the context store or state frame of the user task. It is where the 80386 stores a task's vital information when that task is not running; it is also where the 80386 finds all of the information necessary to restart that task when its turn comes again. When a task lies dormant, its TSS holds all of the information to restart it any time.

A TSS is nothing more than an area of read/write memory, like a small data segment. This segment is not accessible to the user program, however, even at privilege level 0. The space defined within a TSS is accessible to the 80386 only.

2.3.2 Paging support for multitasking

The 80386 processor supports paging. It is implemented in two levels. The 32 bit virtual address is split up into three parts of 10bits, 10bits and 12bits. The most significant 10bits index to a page directory and the next 10bits index to a page table. The last 12bits are used to address the memory within the page. The page size of the processor is constant (=4Kb). The page directory and page table sizes are constant equal to the size of a page.

Base address of the page directory is stored in a system register (CR3). The page table base address is obtained from the page directory entry and page base address is obtained from the page table entry. At any point of time the page directory has to be in the main memory and CR3 should point to it. If the page table or the page pointed to by the virtual address is not present in the main memory, the processor generates a page fault exception and the virtual address which caused the exception is stored in a separate read only register (CR2) in the processor. These two registers mentioned above can be accessed from the privileged mode only.

Using the page fault exception, faulted page or the page table can be restored and the execution of the faulted task can be resumed. This way we can implement the virtual memory along with multitasking in 80386.

2.4 Why we have chosen 80386 ?

80386 is the first processor of iAPX 86 family which has all the features mentioned above. The 80286 processor also supports Protected mode addressing, but it lacks paging and it does not support any separate mode for the 8086 tasks in protected mode. Hence this makes 80286 unsuitable for our multitasking system. There are several other issues which have to be taken into consideration in the actual implementation and are discussed in later chapters.

Chapter 3

Design Overview

In this chapter we discuss the implementation of *fork* for MS-DOS operating system on 80386 and the implementation of fork related system calls for the protected mode tasks. Before that we have to discuss the some of the issues in the design of the kernel.

3.1   Kernel environment

The environment of a task under our kernel is more or less same as that of the process under UNIX [Bach86] [Leff879] kernel. A program is an executable file and a task is an instance of program in execution. The multitasking is implemented in our kernel in a similar manner as that in UNIX, by the fork system call. The tasks interact with the system by invoking predefined procedures called system calls. A new task is created by the fork system call. The task which executes the fork call is called the parent task while the new task is called its child. Each task in the system is identified by a unique integer known as the task identifier. A task may choose to terminate using kill system call. The other system calls supported by our kernel are explained in the chapter on implementation.

3.1.1   System calls in the kernel

The system call instructs the kernel to perform various operations for the calling program and exchange data between the kernel and the task. Tasks also use system calls to communicate with other running tasks in the system. The program can enter the kernel mode by invoking a system call. When the system call is called from the user task the parameters

to the system call are copied from the user stack to the kernel stack by the system call handler.

3.1.2 Modes of task execution

A task in our system can execute in one of the two modes, **kernel mode** or **user mode**. A process normally runs in the user mode and enters the kernel mode through a system call, a trap or an interrupt. The kernel mode supports some privileged instructions like process swapping, accessing some special control registers of 80386, handling interrupts etc. Also kernel maintains some system wide data structures which are accessible only in the kernel mode. Each task's virtual address space can be logically divided into User address space and Kernel address space. The user address space can be accessed in any mode; however, the kernel address space is accessible only when the task runs in the kernel mode. In case of VM86 tasks the execution of the MS-DOS or BIOS services is also carried on in the user mode.

3.2 Virtual memory management

The total virtual space for a task is taken as 4Gb, of which the kernel occupies the high end of virtual address space as shown in the figure 4.1. The remaining part of the virtual address space is available entirely for the user task. Kernel supports two types of tasks namely 80386 protected mode tasks and the VM86 MS-DOS tasks. Provisions are also made for the 80286 processor dependent tasks to run on the system. Depending on the type of the task, size of the task virtual address space varies. Virtual memory is implemented through demand paging. More about this is discussed in the implementation. The portion of the virtual address space below the kernel virtual address space is used for the **U-area**.

3.2.1 U-area of the tasks

Every task has its own U-area. In this virtual address space, only the kernel stack at the time of interrupts and exceptions is stored. There is no fixed size for this area as it is allocated through demand paging. The task's virtual address space does not clash with this area because it is at the high end of the virtual address space. This U-area is also used in the case of protected mode tasks, as the kernel stack, for the kernel entry.

3.2.2 80386 protected mode tasks

When a protected mode task is executed first, the associated executable file is loaded into the main memory by our loader. Thus each task is mapped to some part of the memory known as task virtual address space. This address space for each protected mode task is divided into three logical regions namely text, data and stack. It is assumed that the protected mode applications are compiled else where and are loaded using loader in our system. While compiling under a different operating system, the operating system dependent libraries should not be linked. That is in case of compilation on UNIX we can not link system call libraries, in their place our system call libraries given in the appendix should be linked. Unlike UNIX, the positions of the stack and the heap are interchanged. This is done to decrease the system overhead. This allows us not to bother about the stack growth while the heap is growing in size. This change does not cause any problems to all the normal applications on UNIX. The header format of these files must confirm to COFF format. We have taken UNIX kind of virtual address space because UNIX is so far the most accepted multi-tasking operating system. Our aim was to make the UNIX applications run without much changes to the source code, i.e if the applications were to be linked using our libraries, they should run without any problem.

3.2.3 VM86 MS-DOS tasks

In case of the Virtual 8086 mode tasks, the lowest 1Mb is mapped with MS-DOS and the application. The actual virtual address space of the virtual mode tasks is 0x110000 bytes. The extra 64Kb of the virtual address space above the 1Mb is mapped from 0 to 64Kb of the address space. This is done because the actual virtual address space in the case of 80386 in VM86 mode is 0x110000 bytes. The mapping is given in figure 3.1. The virtual address space above this value up to certain high value is provided for the VM86 tasks kernel level heap. This heap space can be made use of by the applications on MS-DOS for the extra storage through system calls.

3.3 Implementation environment

The Computer Science and Engineering Department of IIT Kanpur has an 80386 based Workstation (SUN-386), and quite a few PC-ATs based on 80386. The operating system

on the SUN-386 is SunOS 4.0.3. The operating system used on the PC-ATs is MS-DOS. In addition to the above, there are twelve SUN-3 workstations. All the machines are connected by a local area network based on IEEE 802.3 (Ethernet) standards.

The SunOS operating system is a heavily enhanced version of the 4.2BSD and 4.3BSD UNIX systems derived from the University of California at Berkely. The SunOS also includes features of AT&T, System V.3 UNIX[Sun88e]. In addition, Sun has made some enhancements of its own like Network File System(NFS).

The compilation of the kernel files into object files is done by the GNU cross compiler. This cross compiler runs on SUN-3 Workstations and generates code for the 80386 based system. The linking of the object files is done using GNU linker. And at the time of linking the virtual address at which the kernel will be loaded is specified. The executable file of the kernel is loaded using a loader which is a MS-DOS program. This loader loads the kernel to an address above 1Mb (LOADER_BASE) and passes control to it.
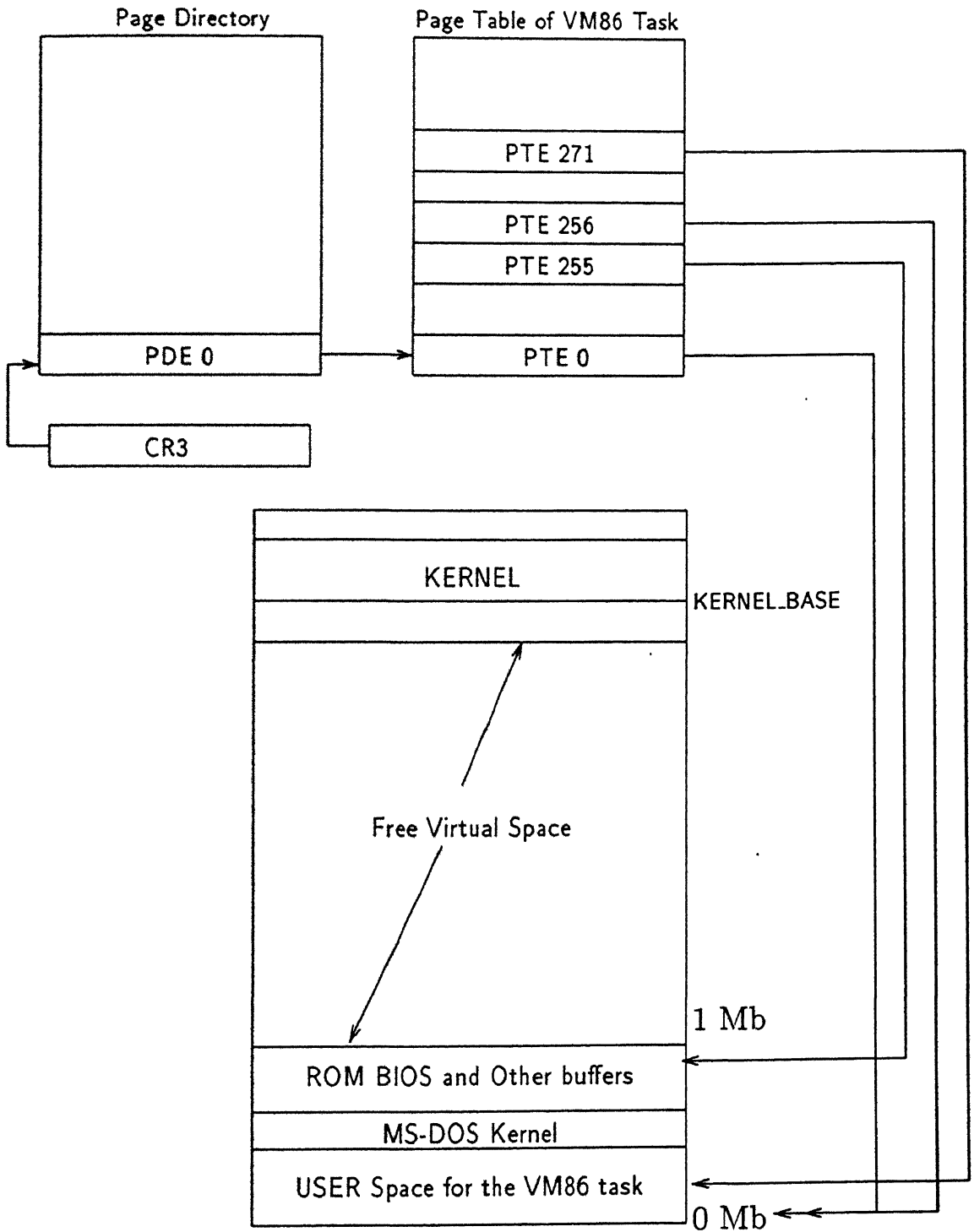
Figure 3.1: Virtual memory mapping of VM86 tasks

Chapter 4

Implementation of Multitasking

The implementation of the multitasking can be divided into the following sections,

1. Entering protected mode and loading of the Kernel.

2. Kernel startup.

3. Preparing the first VM86 task and jumping to it.

4. Interrupt handling.

5. System calls supported by the Kernel.

6. Page fault handler implementation.

4.1   Entering protected mode and loading the kernel

There are two programs in our implementation. The first one is the loader and the second one is the kernel. In this section, we mainly concentrate on our loader implementation. The MS-DOS loader program is used to load the kernel in RAM. The loader starts execution in the real mode of 80386. It loads Kernel program at location LOADER_BASE (= 0x110000). However for this the loader need not enter the protected mode, as MS-DOS provides service for loading data in the extended memory through Int 15h function 0x89. The loading address of the kernel must be same as that of the value specified to the linker at the time of linking. The loader expects the header of the kernel executable file in the COFF [] format.

The loader also assembles the general system information for the kernel startup which contains the following.

1. First task register values.

2. Hard disk parameter tables.

3. Swap space partition for the implementation of the Virtual memory.

4. RAM information like base memory and extended memory.

5. Video information.

6. Other miscellaneous information.

The above information is stored at address LOADER_DATA_BASE (= 0x100000), in the form of a structure. Loader first collects this data from the MS-DOS tables. It loads Kernel program at location LOADER_BASE (=0x110000) (as discussed in the next paragraph).

This information is moved to LOADER_DATA_BASE after entering the protected mode. In order to enter the protected mode, several tables are to be initialized. We initialize Global Descriptor Table (GDT) while executing the loader program. Here no other tables need to be initialized as the interrupts are disabled at the time of passing control to the Kernel.

The GDT descriptors are prepared in the following manner. Code, data and stack segments are mapped in such a way that they map to the real mode code, data and stack segments of the loader itself. Hence after entering the protected mode the code will run as if it is running in the real mode. These segments are declared 16bit segments so that addressing within the segment is same as that in the real mode. Few extra descriptors are also initialized in GDT which are used for the kernel code and the data while passing control to the kernel. GDT also contains two more descriptors, one for data read/write in video memory and the second for general system information data passed to the kernel by the loader. Having initialized the GDT, loader disables the interrupts and enter in the Protected mode. The loader enables the A20 line of the processor to access extended memory. After this, the general system information data is written at LOADER_DATA_BASE location. The loader then passes control to the kernel.

The execution of the loader is not yet complete and the control is not yet passed to MS-DOS. The loader contains a routine vm_task() which exits to MS-DOS. The real mode

address of this routine is passed to the Kernel in the general configuration information. These values are loaded in the first task's TSS (Task State Segment) at the time of task switch by the kernel. Hence after the task switch the control again comes to the loader in VM86 mode to vm_task() routine. The execution of loader is now complete and it exits to MS-DOS in VM86 mode.

4.2 Kernel startup

The kernel startup implementation includes the following.

1. Enable paging

2. Initialization of the kernel data structures

The environment of the kernel when it gets control from the loader will be as follows. No stack is available in the RAM. The processor is in the protected mode. The CS and DS i.e the code and the data segment selectors point to descriptors with base zero and the limit of 4GB. The addressing is 32bit. Also while accessing a data location corresponding to the global variables Kernel need to convert the addresses to the physical RAM addresses because the linker generates all the addresses as virtual and relative to the kernel base. It may be noted that at the time of linking of the kernel code, we provide the virtual base address (= KERNEL_BASE) as the loading address.

4.2.1 Enabling paging

To implement the paging in the complete system, the kernel needs to initialize the following data structures.

- Page directory and the initial page tables

- GDT, LDT, IDT etc.

The page directory and the initial page tables are prepared in the following manner. The physical address of the end of BSS data segment of the kernel is found and it is rounded off to the next page boundary. At this address location page directory is prepared. Which

is used for virtual memory implementation. In our implementation, we use a single page directory available to all tasks.

The initial page tables are prepared at the next page boundaries after the page directory. The initially prepared page tables include the following.

- Page tables (one or more depending on the code size) for mapping the entire kernel, page directory and the initial page tables.

- A page table for accessing the page tables of the tasks. All entries in this are set to zero to indicate the absence of the pages.

- Page tables (one or more depending on the RAM size) for mapping of the main memory (RAM) at some high end of the virtual memory. This page table allows the kernel to access any page of the RAM irrespective of the task to which it belongs.

- A page table for the kernel heap area.

The entries for the kernel, the page directory and pages containing the all the page tables mentioned above are made in the page tables. The entries for these page tables are also made in the page directory simultaneously. Each page of the kernel is mapped to two different addresses, one corresponding to the kernel virtual address space and the second corresponding to the actual physical address. The remaining page tables are not initialized for the start of paging and are set at runtime of the kernel.

After these page tables some pages are allocated for the initial Kernel stack and are freed at the time of the fork of the first task.

The GDT (Global Descriptor Table) and the IDT (Interrupt Descriptor Table) are initialized at the time of compilation of the kernel and are different from those in the loader. These two tables are not altered during the run time of the kernel. The LDT (Local Descriptor Table) for the 386 protected mode tasks is also made ready at the time of compilation of the kernel. VM86 tasks don't need any LDT hence no LDT is provided for these tasks.

After initializing the above data structures paging is enabled. Immediately after the enabling of the paging the kernel code will run at the physical RAM address location. It may be noted the data access can now be done with the virtual addresses or with the real mode addresses. IDTR (Interrupt Descriptor Table Register), GDTR (Global Descriptor

Table Register) and LDTR (Local Descriptor Table Register) are loaded and a jump to the virtual address location is made. At this point the kernel works in the virtual memory space and any C language routine can be called.

4.2.2   Initialization of the kernel data structures

After the initialization of the paging a call is made to K_main() which is the kernel main routine written in C. Here entire virtual space of 4Gb is accessible for the data and the code. However if an address is used which is not mapped in the page tables, it causes system to crash because no exceptions in the kernel can be caught by the exception handlers. This is due to the fact that there is no initialized exception stack. Therefore, the kernel is expected to be free of errors.

The general system data (see section 4.1) passed by the loader is copied into the Kernel structures. Simultaneously other Kernel structures are also initialized.

After this, the kernel data structures are completely initialized and the kernel is ready to run. The stack pages allocated initially are freed now. This freeing of stack pages does not alter the page table entries. The RAM space is mapped at the high end of the virtual space. The kernel virtual space organization is illustrated in figure 4.1.

4.3   Preparing the first VM86 task and jumping to it

The register values which should be loaded in the different registers at the time of first task jump are passed by the loader when it passes control to the kernel. Every task in the system has a **proc** structure associated with it. This structure contains all the data required for the task to run. The fields of the *proc* structure have to be filled for the first task and then task switch is made to it. The fields of the *proc* structure contain the following data.

1. Type of task

2. Amount of virtual space allocated to the task

3. Pointers to its children and its parent

4. Pointers to the page tables of the task

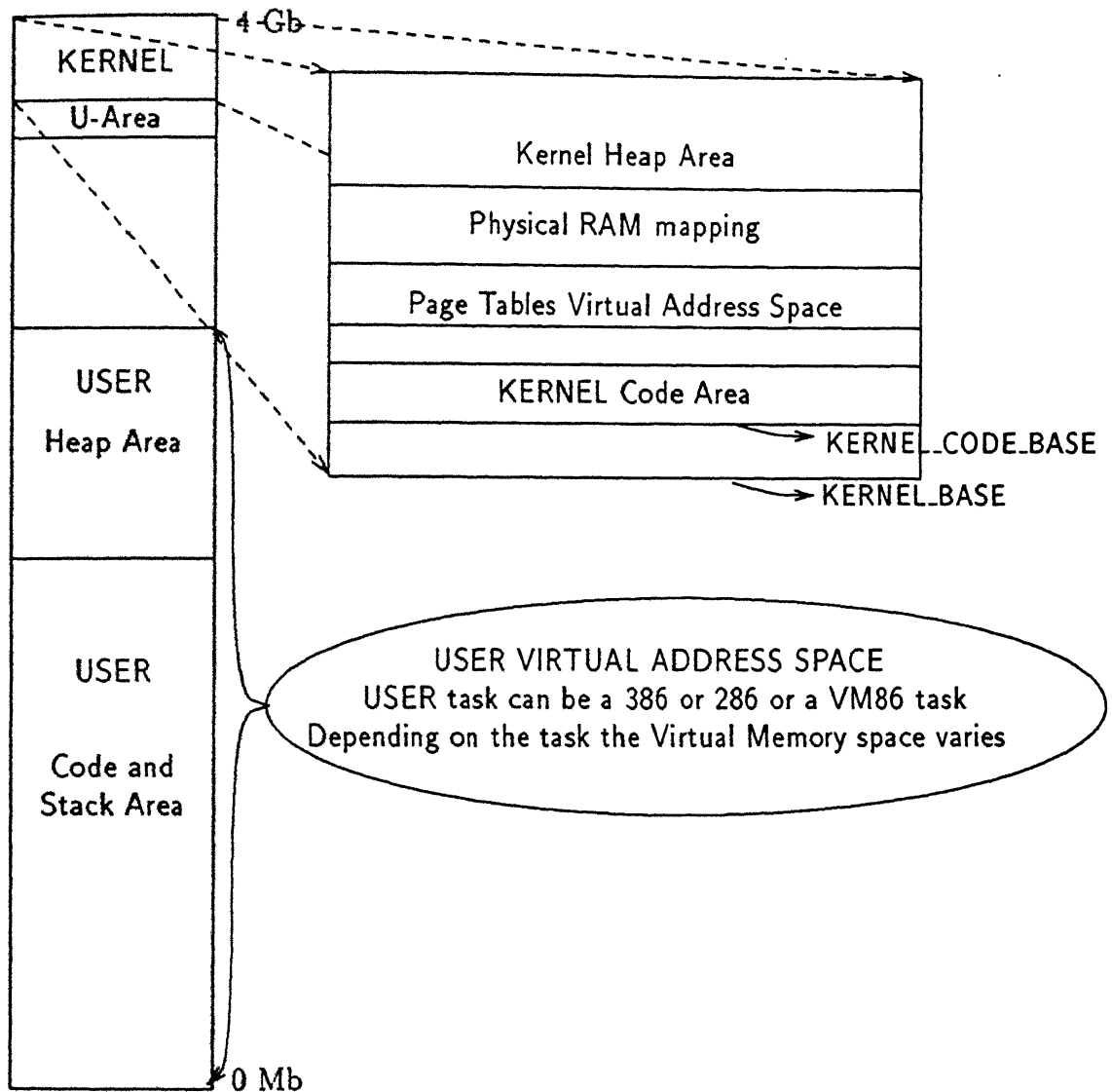5. Task's hardware status i.e the contents of the registers

Figure 4.1: Kernel virtual address space

6. Screen status of the task

7. Heap allocation of the task

8. Resources used by the task

9. Scheduling information

10. And other miscellaneous resources of the task

The *proc* structures of all the tasks are allocated in the kernel heap area.

The general description of the different fields of the *proc* structure are as follows. The type of the task can be T_KERNEL, T_386_TASK, T_286_TASK, T_VM86_TASK or T_DISPATCHER. If it is T_KERNEL it means it is the *proc* structure of the kernel. The other task types are clear. The type of task T_DISPATHER is a VM86 task used for executing MS-DOS calls from the Kernel. The amount of virtual space of any task is the total allocated space of the task in the main memory and the swap area. The pointer to the children and the parent are the pointers to the *proc* structures of the children and the parent. The pointers to the page tables contain a linked list of all allocated page tables of the task. These page tables are entered in the page directory at the time of scheduling this task to the CPU. The screen information contains a buffer to the screen data, whether the task is holding the screen or not, the cursor position information, the screen mode and the buffer size. This information is required at the time of screen swapping with the <sys-req> key or any other system call. The heap allocation of the task is the heap area which is allocated at the time of brk() and sbrk() system calls. The resources used by the task can be hard disk, printer, etc. The kernel does not support any synchronization in the case of the resources. However we provide user level primitives for implementing the synchronization.

The kernel also has a *proc* structure associated with it. Treating the kernel also as one of the tasks simplifies the use of the system call services for the kernel. Thus we don't have to write separate routines for the kernel system calls like brk(), sbrk() etc. Some values of the *proc* structure are not relevant to the kernel and are set to NULL. Page faults can occur only in the kernel heap area and no where else inside the kernel.

Kernel then sets up the *proc* structure for the loader which itself is a VM86 MS-DOS task. An extra page table and some pages are mapped for the U-area (see section 3.2.1) also. For a task scheduled to the CPU, the U-area is always in the main memory. The system halts if the U-area is not resident in the main memory.

While task switching to a task, the registers in the *proc* structure of the task are copied into a general purpose TSS. The task register is loaded with a TSS selector value which points to an unused TSS. Busy bit of the task descriptor in the GDT is reset. Finally the task switch to the VM86 task is made using the new value of TSS selector.

The control comes to a routine in the loader. As said earlier this routine makes an exit to the MS-DOS which causes the cammand prompt to be displayed. A user can now run all MS-DOS applications.

The security of the system is same as in the case of MS-DOS. The user is given all input output permission. The user is not restrained from addressing any input output port of the system. Hence the user can control the input output devices directly and his actions may in turn lead to the crashing the system. We are not guaranteeing any kind of security to the system. What ever harm the user can do under MS-DOS can also be done in our system.

4.4 Interrupt handling

The interrupt handling of VM86 tasks and the protected mode tasks are completely different. All the interrupts enter Kernel through the interrupt gate. When a interrupt occurs the interrupt number is saved on the stack. The type of task from which the interrupt has occurred is determined. Depending on the type of task appropriate interrupt processing is done.

4.4.1 VM86 MS-DOS task interrupts and exceptions

When ever an interrupt or exception occurs in the VM86 mode the following things are done by the hardware automatically.

1. All the data selectors i.e DS, ES, GS and FS are pushed into the kernel stack.

2. The SS and ESP register values of the VM86 task are pushed next into the kernel stack.

3. The EFLAGS, EIP and the CS registers are pushed next.

4. The SS and ESP register values are loaded to point to kernel stack, from task's TSS.

5. The CS and EIP register values are loaded from the IDT entry of the interrupt.

6. The DS, ES, FS and GS are zeroed by the hardware when it enters the kernel.

Depending on the interrupt number the interrupts can be classified into the following three types.

1. MS-DOS/BIOS software interrupts

2. Hardware interrupts

3. 80386 exceptions

The determination as to which type the interrupt belongs requires some processing. Since some of the exceptions clash with the hardware and software interrupts we have to determine to which type the interrupt belongs. The clashes are resolved in the following manner.

The exceptions which pass error codes in the stack are identified by the stack size. The interrupt numbers below 6 can be taken as exceptions or MS-DOS/BIOS software interrupts as the processing is same in both cases i.e redirecting them to MS-DOS/BIOS interrupt service routine. By the above two methods all the exceptions excluding 6, 7, 9 and 16 can be identified. It may be noted the reserved exceptions of 80386 are taken as software interrupts of the VM86 mode itself. These four cases are resolved in the following manner.

The interrupt 6 is taken as 80386 exception i.e invalid opcode. As resolving the clash between the software interrupt and the exception in this case is difficult it is taken as an exception. This assumption can be justified as in the case of 8086 interrupt 6 is reserved hence normal applications should not use this interrupt as a software interrupt. This assumption does not inhibit the transparency of our kernel.

In case of interrupt 7 the following processing is done. The interrupt 7 in the case of 80386 is a fault for coprocessor not present. This exception will occur if the processor encounters a coprocessor instruction and if the coprocessor is not present in the system. Since it is fault, the IP register value in the kernel stack will point to the faulted coprocessor instruction. Using the CS and IP values from the kernel stack the opcode of the faulted instruction is read. If the faulted instruction is a coprocessor instruction, then the IP value is incremented to the next instruction and the control is sent back to the VM86 task itself. If the faulted instruction is not a coprocessor instruction then it is taken as software VM86 task interrupt and it is redirected to the VM86 interrupt service routine itself.

In case of interrupt 9, it can be hardware keyboard interrupt or exception 9. In both the cases the stack frame will be same. To distinguish from the exception or the hardware interrupt we read the interrupt controller to ascertain the fact. If the IRQ1 is set then it is taken as the hardware interrupt otherwise it is taken as an exception.

Interrupt 16 clashes with the video BIOS interrupt and the exception 16 which stands for coprocessor error. The clash is resolved by reading the coprocessor status, if the coprocessor

is present. Otherwise it is taken as BIOS software video interrupt.

MS-DOS/BIOS software interrupts

The general processing of these interrupts involves either redirecting them to the VM86 task's interrupt service routine or ignoring the interrupts. Before taking any action the following is checked.

1. If the interrupt is 0x21 and the function numbers are greater than 0x80 then they are taken as the kernel system calls.

2. If the interrupt is 0x15 and the AL register value is 0x85 then the screen swapping is done.

3. If the interrupt is 0x21 then some of the MS-DOS calls are queued depending on the resource availability.

4. In the case of 0x15 also some of the functions are queued depending on the resource availability.

Hardware interrupts

The actual processing of the hardware interrupts is not done in the Kernel and is left to be handled by the VM86 mode task. This kind of methodology gives greater transparency to the VM86 tasks. The processing of hardware interrupts in the kernel is as follows.

The interrupt 8 (for timer) is used for the task scheduling of the VM86 tasks. The tasks are scheduled in a round robin fashion. The processing required for this interrupt is kept to a minimum. After finishing the kernel processing the control is returned back to the VM86 interrupt service routine.

For the interrupt 9 (for keyboard) is encountered the control is passed to the virtual task holding the screen (foreground task). If the task which is running is not holding the screen then it is preempted and the task which is holding the screen is run.

All the other hardware interrupts are directly redirected to the VM86 task's interrupt service routine. Interrupts are not enabled when ever a hardware interrupt is encountered. They get enabled when the value of EFLAGS is restored from the kernel stack.

80386 exceptions

The following exceptions are redirected to the VM86 task's interrupt service routine.

- Exception 0 (Divide by zero)

- Exception 1 (Debug)

- Exception 2 (NMI)

- Exception 3 (Break Point)

- Exception 4 (Overflow)

- Exception 5 (Bound Error)

In the case of exception 7 (No coprocessor present) the processing is done as described above. In following cases, the VM86 task is terminated and the cause of termination is displayed.

- Exception 6 (Invalid opcode)

- Exception 8 (Double fault)

- Exception 9 (Coprocessor overrun)

- Exception 10 (Invalid TSS)

- Exception 11 (Segment not present)

- Exception 12 (Stack fault)

- Exception 13 (General protection fault)

- Exception 16 (Coprocessor Error)

In case of exception 14 i.e page fault, the control is passed to the page fault handler and after allocating a page in the RAM the control is returned back to the VM86 task. The detailed description is given under page fault handler implementation. All the remaining exceptions which are reserved in the case of 80386 are passed to the VM86 mode task.

4.4.2 Protected mode task interrupts and exceptions

In case of hardware interrupt occurring in the protected mode tasks the Kernel sends an EOI (end of interrupt) to the interrupt controller. And a driver routine is called. As there are no drivers written in our Kernel no processing is done, and a null routine is called.

4.5 System calls supported by the kernel

The following system calls are supported by the Kernel

- check_mult()

- fork()

- kill()

- getpid()

- getppid()

- shutdown()

- block_res()

- free_res()

- Ksbrk()

- Kbrk()

- Kput_dword()

- Kput_word()

- Kget_dword()

- Kget_word()

- Ksend_msg()

- Krecv_msg()

The detailed usage of these system calls is given in the appendix-A. Here we provide the description of each system call. All system calls are implemented in a the uniform manner to the VM86 and the protected mode tasks.

4.5.1 check_mult()

This call is useful for the VM86 task's to check up the presence of the kernel. If the kernel is present it returns a value of 0 otherwise it is undefined and is non-zero. The kernel sets NULL value to the AX register and the control is passed back to the VM86 task itself.

4.5.2 fork()

This system call is similar to the one in UNIX [Bach86] [Leff89]. In this system call a child task is created by copying the virtual space of the parent into the child's virtual space. Only difference between the parent and the child is the value of return value returned to the parent and the child. To the parent the task id of the child is returned and to the child zero value is returned.

4.5.3 kill()

This system call is used to terminate a task. The Kernel displays the termination message and terminates the task.

4.5.4 getpid()

This call returns the task's task-id. Every task has a task id associated with it and it is stored in the *proc* structure.

4.5.5 getppid()

This call returns the task's parent id. Each user task in the system has a parent. The kernel will be the parent of tasks which do not have any user task as parent.

4.5.6 shutdown()

This call is used to terminate the multitasking Kernel.

4.5.7   block_res()

This call is used for blocking a resource. This helps the tasks in synchronize among themselves. Through this call the resource number is sent to the kernel to block the resource if it is not already blocked. If the resource is already in use then the tasks requesting the resource will be blocked till it is freed.

4.5.8   free_res()

This call is used for freeing a resource.

4.5.9   Ksbrk()

This call is used to increase or decrease the task's heap space.

4.5.10   Kbrk()

This call is used to increase or decrease the task's heap space and is similar to Ksbrk() call.

4.5.11   Kput_dword()

This call puts a double word into the task's heap space. This is useful in the case of VM86 tasks as the virtual space is limited to 1Mb and hence these tasks can not access the heap space directly.

4.5.12   Kput_word()

This call puts a word into the task's heap space. This is useful in the case of VM86 tasks for the above stated reason.

4.5.13   Kget_word()

This call is used to get a word from the task's heap space. This is useful in the case of VM86 tasks for the above stated reason.

4.5.14   Kget_dword()

This call is used to get a double word from the task's heap space. This is useful in the case of VM86 tasks for the above stated reason.

4.5.15 Ksend_msg()

This routine is used for message passing from one task to another. The kernel copies the message from the user space to the kernel space and attaches a header specifying the task to which the message is it has to be sent. The message will be kept in the kernel buffer till the other task requests for the message through Krecv_msg(). The ordering of the messages is strictly maintained with the help of a linked list.

4.5.16 Krecv_msg()

This routine is used by a task to receive a message previously sent by another task using Ksend_msg() call.

4.6 Page fault handler implementation

The 80386 supports built in paging mechanism as described in chapter 2. Whenever a virtual address is generated by the processor for which the corresponding page is not present in the main memory, a page fault exception is raised by the processor.

Once the control comes to the page fault handler minimum assumptions regarding the segmentation are assumed. The faulted virtual address of the page is read, from the CR2 register. By reading the entries of the page directory and the page table pointed by the virtual address the reason for the page fault is ascertained. The page fault handler is written to take care of the addresses for which the page table may be not present in memory. The error code passed by the page fault exception is also used to find the cause of the page fault. The cause of the page fault can be one of the following reasons.

1. Page pointed by the virtual address is not present in the RAM.

2. Page is present in the RAM but it is read only.

3. Page table of the virtual address is not present in the RAM.

The first two reasons can be attributed to the commonly occurring page faults. The third reason should not occur in the present design as the page tables of tasks are never swapped. In case of page fault due to the first reason, after mapping the faulted page to the main memory the control is returned to the task. In other cases the task is terminated.

4.6.1 Swap area

A partition of the disk space is used for the swap area. The information about the swap area is stored at the time of the initialization. The whole swap area is divided into logical pages. Every logical page number is directly mapped to the disk address. The low level mapping of the logical page number to the disk address is completely transparent to the page fault handler. The page fault handler deals at the level of logical page numbers in the swap area.

The Kernel uses a built in hard disk device driver to read and write from any random sector of the hard disk.

The entire pages of the main memory (RAM) and the swap area have bitmaps specifying the status of each page in the RAM and the swap area. A bit set in any of the two bitmaps indicates that the particular page is in use. These structures are used to get the free pages in the RAM and the swap area.

There is another array used in the implementation of the page fault handler. Each entry of the array contains the following fields.

- pointer to the page table entry

- counter for each page

The number of entries of this array will be same as that of the number of pages in the RAM. The field counter is used for the implementation of the LRU (Least Recently Used) algorithm [Pet85] on the pages in the RAM. The least recently used page is found for the making space in the RAM.

Once the faulted page is determined, the page fault handler gets a free page from the RAM. If the free page is not available then space is made in the RAM by using the LRU algorithm. Then the page table entries and the other data structures required for the implementation of the paging are changed and the control is sent back to the task. The implementation of the page fault handler is independent of type of the task.
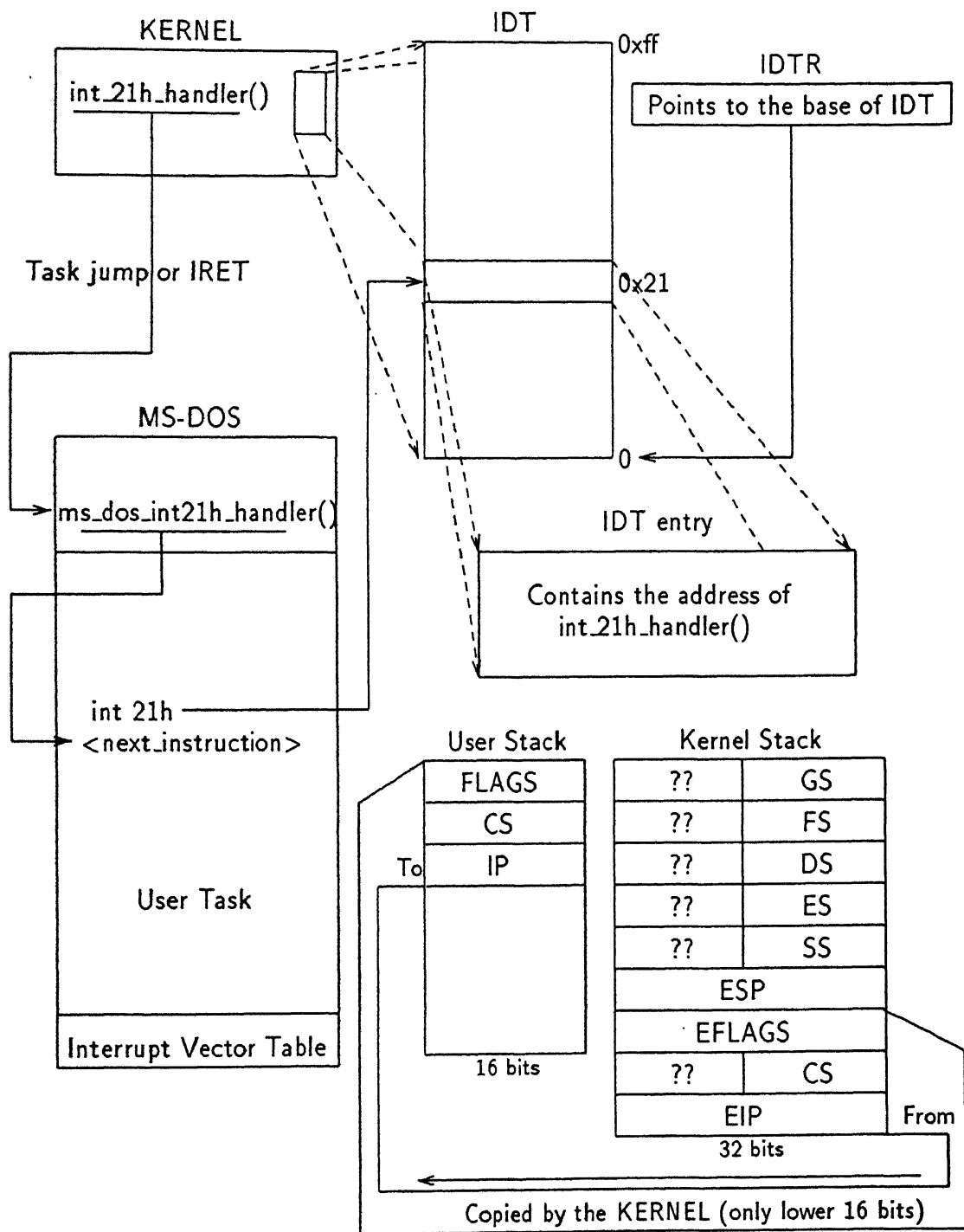
Figure 4.2: Interrupt and exception handling for VM86 tasks

Chapter 5

Conclusion and future additions

We had started to write a small Kernel platform for implementing X - Windows and other relevant software on PC-386ATs. The project got struck at some points during the debugging of low level machine interface. Due to the lack of detailed literature about 80386 lot of time got consumed in debugging the software. We learned quite a lot about the processor while developing the software. Lot of time was lost while writing the 'printf' kind of routines for the debugging.

As we wrote the kernel from the scratch no existing debuggers could be used for debugging the Kernel code as it involved the debugging through various modes of the processor. This made our task very tough. We also faced some compilation problems. As there is no MS-DOS based 32bit compiler this forced us to compile our entire Kernel using GNU C UNIX compiler. The linking of assembly and C linking are not properly documented in any manual. Worst still the problems surface at run time that to only after entering the protected mode.

Initially we started with an idea of providing multitasking support for only VM86 tasks. This gave us the idea of writing the entire Kernel as a COM program in the MS-DOS itself, but once our Kernel complexity started increasing we could not accommodate the entire code in a single COM file. Since some applications code may exceed 1Mb virtual space limitation, we implemented 'fork' for protected mode as well.

The other problem we faced was lack of information about MS-DOS. The user level information available in abundance was not enough. Some undocumented software interrupts of MS-DOS created lot of problems.

Following general mistakes committed while developing the Kernel were, the following guidelines can be outlined for system programmers.

- In the case of VM86 tasks we should be careful about the stack wrap round after 64Kb. Supposing if the SP (Stack Pointer) of the VM86 task holds a value 0 and we push a 16bit quantity into the stack it will be moved to the location SS:[0xfffe].

- In the case of loader writing and compiling the entire code in 'Tiny' or 'Small' models of compilation is far easier. We have to be very careful while writing the assembler directives for these models of compilation.

- While copying the FLAGS value from the Kernel stack to the VM86 task's stack be sure of resetting the IF and TF bits.

5.1   Summary and future improvements

We have implemented 'fork' call complete in all respects. The existing inter process communication is very basic. We thought of including BSD Sockets and UNIX kind of signals mechanism in our Kernel but due to lack of time we could not achieve those objectives. The following improvements can be made to our Kernel to run the UNIX based software on our Kernel.

- Implement BSD Sockets. For this purpose the existing BSD UNIX source code can be used. The low level ethernet driver code and all the other necessary code is available, only it has to be extracted from it and has to be compiled with our Kernel by providing extra system calls.

- Implementation of Signals. For implementing the signals the BSD UNIX code can not be used. Since our Kernel supports basic system calls for the multitasking adding extra features should not be difficult.

- Other miscellaneous system calls can also be implemented depending on the application one wishes to run on the PCs.

With the above improvements most of the UNIX based software can be run on the PCs.

Bibliography

[Bach86] Bach,M.J., *The Design of the UNIX Operating System*, Prentice-Hall Inc., Engelwood Cliffs, May 1986.

[Leff89] Leffler,S.J., McKusick,M.J., Karels,M.J., Quarterman,J.S., *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley Publishing Co., May 1989.

[Pet85]   Peterson,J.L, and Silberschatz,A., *Operating System Concepts*, Addison-Wesley Publishing Co., 1985

[Turl88] Turley,J.L., *Advanced 80386 programming techiniques*, Osborne McGraw-Hill 1988

Appendix A

Usage of the system calls

All the system call's parameters are 32bit quantities. This is done for the sake of uniformity of VM86 and protected mode tasks. The return value of the system call can be a 32bit or a 16bit quantity. The following assumptions are made about the compiler default types.

The return value will be returned in "AX" or "EAX" depending on the size of the return value i.e 16bit or 32bit.

A.1 Virtual 8086 tasks

In case of VM86 task system calls they are supported in the following manner. They are implemented as extensions to MS-DOS system calls i.e int 21h services. Our kernel services can be invoked through int 21h and AH = 0x80. The "AL" register value identifies the type of the system call. The value of the AL register for different system calls is mentioned along with their description. The parameters to the system call are copied from the user stack of the task.

- The word length is taken as 2 bytes or 16bits.

- Stack is aligned to the word boundary.

- The size of "integer" of "int" is taken as 16bits.

- The size of "long" is taken as 32bits.

- The size of near pointer is taken as 16bits.

- The size of far pointer is taken as 32bits or 4 bytes.

- All the system calls are made through a far call.

A.2   Protected mode 80386 tasks

A call gate is used to enter kerner for the system call services. The "AL" register value identifies the type of system call. The parameters to the system call are copied from the user stack of the task.

- The word length is taken as 4 bytes or 32bits.

- Stack is aligned to the word boundary.

- The size of "integer" or "int" is taken as 32bits.

- The size of "long" is taken as 32bits.

- The size of "short" is taken as 16bits.

- The size of near pointer is taken as 32bits.

- The size of far pointer is taken as 48bits or 6 bytes.

- All the system calls are through a near call.

A.3   System call check_mult()

NAME

check_mult - Checks the presence of multitasking Kernel

SYNOPSIS

VM86 Tasks

int far check_mult(void);

Protected Mode 80386 Tasks

int check_mult(void);

DESCRIPTION

check_mult() - checks the presence of the multitasking Kernel. This is useful for the MS-DOS VM86 tasks. This routine makes it easy to write portable software on MS-DOS. The

"AL" register value = 0x0

RETURN VALUE

check_mult() returns zero if multitasking is present, otherwise the return value is undefined.

A.4   System call fork()

NAME

fork - creates a child task

SYNOPSIS

VM86 Tasks

int far fork(void);

Protected Mode 80386 Tasks

int fork(void);

DESCRIPTION

fork() creates a new task. The new task i.e the child task is a exact copy of the calling task except for the following:

- The child has a different and unique task id.

- Both the child and the parent tasks have different parents.

- The resource utilization is zero for the child.

The "AL" register value = 0x1

RETURN VALUE

fork() returns the task id of the child to the parent and zero value to the child upon successful completion. Upon failure it returns -1.

A.5 System call getpid()

NAME

getpid - gets the task id of the task.

SYNOPSIS

VM86 Tasks

int far getpid(void);

Protected Mode 80386 Tasks

int getpid(void);

DESCRIPTION

getpid() - This call returns the task id of the task. This integer value is unique to each task. The "AL" register value = 0x2

RETURN VALUE

This call returns the task id of the task. There is no error value returned from this call.

A.6 System call getppid()

NAME

getppid - gets the task's parent id.

SYNOPSIS

VM86 Tasks

int far getppid(void);

Protected Mode 80386 Tasks

int getppid(void);

DESCRIPTION

getppid() - This call returns the task id of the parent the calling task. The "AL" register

value = 0x3

RETURN VALUE

This call returns the parent id of the task. There is no error value returned by this call.

A.7 System call shutdown()

NAME

shutdown - Terminates the multitasking.

SYNOPSIS

VM86 Tasks

int far shutdown(void);

Protected Mode 80386 Tasks

int shutdown(void);

DESCRIPTION

shutdown() - This call terminates the multitasking in the system and enters an infinite loop displaying the message for rebooting the system. The "AL" register value = 0x4

RETURN VALUE

This call does not return to the user.

A.8 System call kill()

NAME

kill - Terminates a task unconditionally.

SYNOPSIS

VM86 Tasks

int far kill(long task_id);

Protected Mode 80386 Tasks

int kill(int task_id);

DESCRIPTION

kill() - This call terminates a task unconditionally. The parent can kill any of its children. But the child can not kill its parent. If a parent is killed then its children are adopted by the parent's parent. The "AL" register value = 0x5

RETURN VALUE

This call returns 0 on success and -1 on failure.

A.9 System call block_res()

NAME

block_res - Blocks a resource of the system.

SYNOPSIS

VM86 Tasks

int far block_res(long resource_id);

Protected Mode 80386 Tasks

int block_res(int resource_id);

DESCRIPTION

block_res() - This call blocks a resource of the system. If the resource is already blocked the task is kept waiting for the resource to be freed. If the time of waiting elapses then it returns error value. The resource numbers are given in the Appendix B. The "AL" register

value = 0x6

RETURN VALUE

This call returns 0 on success and -1 on failure.

A.10   System call free_res()

NAME

free_res - Frees a resource of the system.

SYNOPSIS

VM86 Tasks

int far free_res(long resource_id);

Protected Mode 80386 Tasks

int free_res(int resource_id);

DESCRIPTION

free_res() - This call frees a resource of the system. The "AL" register value = 0x7

RETURN VALUE

This call returns 0 on success and -1 on failure.

A.11   System call Ksbrk()

NAME

Ksbrk - Changes the heap size.

SYNOPSIS

VM86 Tasks

long far Ksbrk(long incr);

Protected Mode 80386 Tasks

char * Ksbrk(int incr);

DESCRIPTION

Ksbrk() - This call changes the heap allocation for a task. In case of VM86 tasks also can use this function for the extra storage. In this function *incr* more bytes are added to the program's heap space and a pointer to the start of the new area is returned. The value of incr can be +ve, -ve or 0. The "AL" register value = 0x8

RETURN VALUE

It returns the old break value on success and on failure it returns -1.

A.12 System call Kbrk()

NAME

Kbrk - Set the break address.

SYNOPSIS

VM86 Tasks

long far Kbrk(long addr);

Protected Mode 80386 Tasks

int Kbrk(char *addr);

DESCRIPTION

Kbrk() - This call sets the heap last used address for a task. In case of VM86 tasks also can use this function for the extra storage. In this function *addr* is set to the address of the break value. This is just an alternate function compared to that of Ksbrk(). The "AL"

register value = 0x9

RETURN VALUE

This call returns 0 on success and -1 on failure.

A.13 System calls for accessing heap

NAME

Kput_dword, Kput_word, Kget_dword and Kget_word -

SYNOPSIS

VM86 Tasks

int far Kput_dword(long);

int far Kput_word(long);

int far Kget_word(long);

int far Kget_dword(long);

Protected Mode 80386 Tasks

int Kput_dword(long);

int Kput_word(long);

int Kget_word(long);

int Kget_dword(long);

DESCRIPTION

Kput_dword(), Kput_word(), Kget_dword(), Kget_word() - These routines as their implies are used for accessing the data items from the task's heap area. The "AL" register values

are = 0xa, 0xb, 0xc and 0xd respectively.

RETURN VALUE

All the *get* functions return the value got from the heap area and the *put* kind of functions return value is not relevant.

A.14 System calls for message passing

NAME

Ksend_msg, Krecv_msg - To receive and send messages between the tasks.

SYNOPSIS

VM86 Tasks

int far Ksend_msg(long task_id, char far *addr, long size);

int far Krecv_msg(long task_id, char far *addr, long size);

Protected Mode 80386 Tasks

int Ksend_msg(int task_id, char *addr, int size);

int Krecv_msg(int task_id, char *addr, int size);

DESCRIPTION

Ksend_msg() - This call is used for sending messages to another task. The task_id identifies the other task. The buffer and its size are given in the other parameters.

Krecv_msg() - The parameters have the same meaning as that of the previous call except that this system call is used for receiving messages.

In either of the cases if the task mentioned is not active then the calls will fail. If the task is active then Ksend_msg() will return immediately where as Krecv_msg() will be blocked till the message is received. The "AL" register values are = 0xe and 0xf respectively.

RETURN VALUE

These calls returns 0 on success and -1 on failure.

Appendix B

Resource numbers

The resources of a general system are the following.

```
typedef enum
{
        FDISK1_REQUIRED,
        FDISK2_REQUIRED,
        FLOPPY1_REQUIRED,
        FLOPPY2_REQUIRED,
        CATRIG_REQUIRED,
        COM1_REQUIRED,
        COM2_REQUIRED,
        COM3_REQUIRED,
        COM4_REQUIRED,
        LP1_REQUIRED,
        LP2_REQUIRED,
        LP3_REQUIRED,
        LP4_REQUIRED,
        RAM_PAGE_REQUIRED,
        SWAP_PAGE_REQUIRED,
        SCREEN_REQUIRED,
```

```
};
```